UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

AD-A205 655

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Data General Corporation, ADE, Version 3.00 MV/20000 Host and Target	5. TYPE OF REPORT & PERIOD COVERED 27 May 1988 to 27 May 1988
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) NBS Gaithersburg, MD	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS NBS Gaithersburg, MD	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) NBS Gaithersburg, MD	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

DTIC ELECTE MAR 0 2 1989 S D H

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ADE, Version 3.00, Data General Corporation, NBS, Gaithersburg, MD, MV/20000 under AOS/VS, Version 7.56 (Host) to MV/20000 under AOS/VS, Version 7.56 and MV/15000 Model 20 under AOS/RT32, Version 4.03 (Target), ACVC 1.9.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

Compiler Name: ADE, Version 3.00

Certificate Number: 880527S1.09112

Host:	Target:
MV/20000 under AOS/VS, Version 7.56	MV/20000 under AOS/VS, Version 7.56 and MV/15000 Model 20 under AOS/RT32, Version 4.03

Testing Completed May 27, 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Bureau of Standards
Gaithersburg, MD 20899

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria, VA 22311

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

AVF Control Number: NBS88VDGC515

Ada Compiler
VALIDATION SUMMARY REPORT:
Certificate Number: 880527S1.09112
Data General Corporation
ADE, Version 3.00
MV/20000

Completion of On-Site Testing:
May 27, 1988

Prepared By:
Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

This information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of test are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

> To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
>
> To attempt to identify any unsupported language constructs required by the Ada Standard
>
> To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the National Bureau of Standards according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was completed May 27, 1988, at Data General Corporation, Research Triangle Park, North Carolina.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> Software Standards Validation Group
> Institute for Computer Sciences and Technology
> National Bureau of Standards
> Building 225, Room A266
> Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines. Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide., December 1986.

1.4 DEFINITION OF TERMS

ACVC — The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada Commentary — An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard — ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant — The agency requesting validation.

AVF — The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.

AVO — The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler — A processor for the Ada language. In the context of

this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Language Maintenance	The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and Panel possible changes to the ANSI/MIL-STD for Ada.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	An Ada program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that

reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These

tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: ADE, Version 3.00

ACVC Version: 1.9

Certificate Number: 880527S1.09112

Host Computer:

- Machine: MV/20000
- Operating System: AOS/VS, Version 7.56
- Memory Size: 64 MB

Target Computers:

- Machine: MV/20000
- Operating System: AOS/VS, Version 7.56
- Memory Size: 64 MB
- Machine: MV/15000 Model 20
- Operating System: AOS/RT32, Version 4.03
- Memory Size: 24 MB

Communications Network: not used

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See test D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 32 bit integer calculations. (See tests D4A002A and D4A004A.)

- Predefined types.

  This implementation supports the additional predefined types SHORT_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001BC and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises a NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently all default initialization expressions or record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real [illegible] apparently round away from zero. (See test [illegible]

- [illegible]

[illegible] is allowed to raise NUMERIC_ERROR or [illegible] for an array having a 'LENGTH that exceeds [illegible] and/or SYSTEM.MAX_INT. For this [illegible]

[illegible] an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test [illegible])

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises STORAGE_ERROR when the array objects are declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises STORAGE_ERROR when the array objects are declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications during compilation. (See test E38104A.)

In assigning record types with disciminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

  An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is not supported, then the implementation must reject it.

  Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

  Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

  Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

  Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

  Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

  Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

  Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

  Record representation clauses are not supported. (See test A39005G.)

  Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

- Pragmas.

  The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

  The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

There are strings which are illegal external file names for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102C and CE2102H.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for (SEQUENTIAL_IO and DIRECT_IO). (See tests CE2106A and CE2106B.)

Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A..5 (5 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107F..I (4 tests), CE2110B, and CE2111H.)

An external file associated with more than one internal file can be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110[illegible].)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies cannot compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies cannot be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits cannot be compiled in separate compilations. (See test CA3011A, LA5008M and LA5008N.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tests, 28 tests had been withdrawn because of test errors. The AVF determined that 257 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. Modifications to the code, processing, or grading for 2 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	107	1046	1612	15	13	44	2837
Inapplicable	3	5	241	2	4	2	257
Withdrawn	3	2	21	0	2	0	28
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	186	499	528	245	165	98	142	326	131	36	232	3	246	2837
Inapplicable	18	73	146	3	0	0	1	1	6	0	2	0	7	257
Withdrawn	2	14	3	0	1	1	2	0	0	0	2	1	2	28
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 28 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35904B	C35A03E	C35A03R	C37213H	C37213J	C37215C
C37215E	C37215G	C37215H	C38102C	C41402A	C45332A
C45614C	E66001D	A74106C	C85018B	C87B04B	CC1311B
BC3105A	AD1A01A	CE2401H	CE3208A		

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 257 tests were inapplicable for the reasons indicated:

C24113H..K (4 tests) contain lines of source code that are longer than 120, which is the limit for this implementation.

C35702A uses SHORT_FLOAT which is not supported by this implementation.

A39005G uses a record representation clause which is not supported by this compiler.

The following (13) tests use LONG_INTEGER, which is not supported by this compiler.

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

C45531I, C45531J, C45532I, and C45532J use fine 32-bit fixed-point base types which are not supported by this compiler.

C45531K, C45531L, C45532K, and C45532L use coarse 32-bit fixed-point base types which are not supported by this compiler.

C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

C4A013B uses a static value that is outside the range of the most accurate floating-point base type. The declaration was rejected at compile time.

D4A002B and D4A004B use 64-bit integer calculations which are not supported by this compiler.

B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

CA2009F and CA1012A compile generic subprogram declarations and bodies in separate compilations. This compiler requires that generic subprogram declarations and bodies be in a single compilation.

CA2009C, BC3204C, and BC3205D compile generic package specifications and bodies in separate compilations. This compiler requires that generic package specifications and bodies be in a single compilation.

CA3011A and LA5008M..N (2 tests) compile generic unit bodies and

subunits in separate compilations. This compiler requires that generic unit bodies and their subunits be in a single compilation.

AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

CE3111B requires that one (internal) file be able to read a value that was just written by another file that shares the same external file; but this implementation raises END_ERROR, since the value has not yet been flushed from the writing file's buffer. The AVO ruled that this is acceptable behavior.

The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for one (1) Class B test and one (1) Class C test.

BA1101C was modified by commenting out "USE BA1101C0" in order to prevent a semantic error in the compilation of BA1101C5 because BA1101C3's body was not added to the program library.

C4A012B had a NUMERIC_ERROR handler added to the source code as this is an acceptable option which this implementation utilizes.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the ADE Version 3.0 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the ADE Version 3.0 using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a MV/20000 operating under AOS/VS, Version 7.56, and a MV/20000 target operating under AOS/VS, Version 7.56 as well as a MV/15000 Model 20 target operating under AOS/RT32, Version 4.03. The host and target computers were linked via magnetic tape.

A magnetic tape containing all tests was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the tape. Tests requiring modifications during the prevalidation testing were not included in their modified form on the tape. The contents of the tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the MV/20000, and all executable tests were run on the MV/20000 and MV/15000 Model 20. Results were printed from the target computer.

The compiler was tested using command scripts provided by Data General and reviewed by the validation team. The compiler was tested using all default switch settings without exceptions.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and two target computers. Test output, compilation listings, and job logs were captured on magnetic tape and hardcopy and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Data General, Research Triangle Park, North Carolina and was completed on May 27, 1988.

Appendix A

DECLARATION OF CONFORMANCE

Compiler Implementer: Data General Corporation
Ada Validation Facility: The Software Standards Validation Group
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name:	ADE	Revision: 3.00
Host Architecture - ISA:	MV 20000	OS&VER #: AOS/VS 7.56
Target Architecture - ISA:	MV 20000	OS&VER #: AOS/VS 7.56
	MV 15000 Mod 20	OS&VER #: AOS/RT32 4.03

Derived Compiler Registration

Derived Compiler Name:	ADE	Revision: 3.00
Host Architecture - ISA:	MV Family	OS&VER #: AOS/VS 7.56
Target Architecture - ISA:	MV Family	OS&VER #: AOS/VS 7.56
		OS&VER #: AOS/RT32 4.03

Implementer's Declaration

I, the undersigned, representing Data General Corporation have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that Data General Corporation is the owner of record of the Ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

Signature — 6/13/88
Date

Owner's Declaration

I, the undersigned, representing Data General Corporation take full reponsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Signature — 6/13/88
Date

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the ADE, Version 3.0, are described in the following sections which discuss topics in Appendix F of the Ada Standard. Implementation- specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is


     type INTEGER is range -2147483648 .. 2147483627;
     type SHORT_INTEGER is range -32768 ..  32767 ;

     type FLOAT is digits 6 range -7.23700**(+75) .. 7.23700**(+75)
     type LONG_FLOAT is digits 15 range -7.32700557733226**(+75) ..
                                         7.32700557733226**(+75)
     type DURATION is delta 2.0**(-9) range -2**(+22) .. 2**(+22)


end STANDARD;
```

APPENDIX F OF THE Ada STANDARD

ADA PROGRAMMING LANGUAGE REFERENCE MANUAL
ANSI/MIL-STD-1815A

APPENDIX F:

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

ROLM/DGC Ada is a validated implementation of Ada that conforms to the full ANSI/MIL-STD-1815A standard. That standard allows individual implementations to set or define certain language characteristics, such as pragmas, restrictions on representations clauses, capacity limits, and so forth.

This appendix describes the language characteristics which are set or defined by the ROLM/DGC implementation. The appendix contains these sections:

ADE-DEFINED PRAGMAS

This section lists and describes pragma statements implemented in the ADE.

Introduction

Pragmas allow you to exercise some control over how the compiler processes your programs. Pragmas have no effect on the semantics of a program. You use pragma statements to instruct the compiler how to process your programs at compilation time.

Several pragmas are defined in the Ada Language Reference Manual. Other pragmas are defined by the various implementations of the language.

The Ada Language Reference Manual (LRM) describes the language defined standard pragmas and their use. This section provides additional information on those standard pragmas and defines the pragmas that are unique to ROLM/DGC Ada.

This section contains two parts. The first part lists pragmas that are not implemented in the current version of the ADE. The second part lists the implemented pragmas in alphabetical order, describes them and provides examples of their use.

Pragmas with No Effect

The following language-defined pragmas have no effect on compilations in the current version of the ADE:

Pragma	Explanation
CONTROLLED	Because the compiler does not reclaim unused storage automatically, this pragma is not needed. To deallocate storage explicitly, use the generic procedure UNCHECKED_DEALLOCATION. See LRM, section 13.10, and the "ADE User's Manual," chapter 6.
MEMORY_SIZE	The package SYSTEM defines the MEMORY_SIZE constant as 2 ** 29 words. You can adjust the maximum virtual memory size by specifying the /MTOP=n switch on the ADALINK command. See "ADE User's Manual", chapter 4.

Pragmas with No Effect (continued)

OPTIMIZE	The compiler does not currently use TIME or SPACE optimization criteria.
PACK	This pragma has no effect in the current version.
SHARED	The current version does not implement indivisible direct read and update operations for any object. Therefore, there are no objects to which you can apply this pragma. See LRM, section 9.11.
STORAGE_UNIT	The package SYSTEM currently defines the storage unit as a 16-bit word. You can not redefine it in the current version.
SYSTEM_NAME	The package SYSTEM defines this as an object of enumeration type NAME, for which there is only one literal allowed in the current version.

Pragmas Implemented in the ADE

The following pages list in alphabetical order all the pragmas you can use in the ADE.

pragma ELABORATE

Specifies which library unit bodies (secondary units) to elaborate before the current compilation unit.

Format: pragma ELABORATE (library_unit [,library_unit]);

library_unit is the simple name of the library unit whose body you want elaborated before the current compilation unit.

Description

Pragma ELABORATE directs the compiler to cause the body of the specified library unit or units to be elaborated before elaborating the current compilation unit. If the current compilation unit is a subunit, pragma ELABORATE directs the compiler to cause the body of the specified library unit to be elaborated before elaborating the library unit which is the ancestor of the current compilation subunit.

Pragma ELABORATE must appear after the context clause for the current compilation unit and it must specify a library unit named in that context clause. Furthermore, the specified library unit must have a body.

Example:

```
with EARTH_DATA;
pragma ELABORATE (EARTH_DATA);
procedure SOLAR_SYSTEM is
    ...
    EARTH_DATA.TRACK_ORBIT;
    ...
end SOLAR_SYSTEM;
```

For more information see LRM, section 10.5.

pragma ENTRY_POINT

Associates an Ada subprogram name with a specific entry point label so other language routines can call or be called by Ada subprograms.

Format: pragma ENTRY_POINT (subprogram_name, "entry_point_name");

subprogram_name — is the unique name of an Ada subprogram defined in the declarative part of the current compilation unit. Do not use dot notation to specify subprogram_name.

entry_point_name — is the STRING literal denoting the actual external label. Use uppercase letters enclosed in quotes; for example, "FRTN_LIBNAME".

Description

You can use this pragma in either of two ways:

1. A subprogram written in another language can refer to an Ada subprogram using a labeled entry point defined by this pragma.

2. An Ada subprogram can specify a library routine written in another language by giving the name of the routine as an entry point. Pragma ENTRY_POINT is used with pragma INTERFACE in this case.

Pragma ENTRY_POINT must appear in the declarative part of a block, in a package specification, or after a compilation unit. You must specify both arguments.

Example:

```
procedure MAIN is
     function FRTN_OP (X: INTEGER) return BOOLEAN;
     pragma INTERFACE (F77, FRTN_OP);
     pragma ENTRY_POINT (FRTN_OP, "FRTN_LIBNAME");
     ...
begin
...
end MAIN;
```

---

pragma INLINE

Specifies the subprogram and/or generic subprograms that you want expanded inline at each call whenever possible.

Format: pragma INLINE (name [, name])

name — is the subprogram or generic unit you want inlined at each call; name must be a preceding declarative item in this declarative part.

Description

Pragma INLINE directs the compiler to insert code for the body of the named subprogram at each occurrence of a call to that sub-program. If the named subprogram is a generic unit, the compiler inserts code for the bodies of all subprograms that are instantiations of that generic unit.

The following restrictions apply to pragma INLINE:

- The nesting level of inlined procedures cannot exceed 100.
- Inlining is not allowed for
  - . Subprograms that call themselves recursively;
  - . Subprograms containing exception handlers;
  - . Any unit that declares a task, task type, or access to a task type.
- A program is erroneous if it inlines a function that returns an unconstrained object.

Example:

```
procedure WITH_INLINE is
   FIRST, SECOND : INTEGER;
   function SQUARE (S : INTEGER) return INTEGER;
   pragma INLINE (SQUARE);                          <-- pragma
                                                        applies
   function SQUARE (S : INTEGER) return INTEGER is      to all
   begin                                                calls of
      return S * S;                                     SQUARE
   end SQUARE;                                          in MAIN.
begin
   FIRST := SQUARE (2);
   SECOND := SQUARE (SQUARE (FIRST));
end WITH_INLINE;
```

Pragma INLINE (example continued)

The following shows how the compiler obeys the INLINE directive. We compare the assembly (.SR) files produced by the compiler after it processes these two source files:

WITH_INLINE.ADA -- the same example source as above;

WITHOUT_INLINE.ADA -- the same source, but without the pragma.

.SR Instructions Corresponding to WITH_INLINE.ADA

```
;; begin
;; FIRST := SQUARE (2);
;; S : constant INTEGER := 2;
;; return S * S;                              <-- 1st inline expansion
        NLDAI    4,0
        XWSTA    0,12.,3           ;; FIRST

;; SECOND := SQUARE (SQUARE (FIRST));
;; S : constant INTEGER := SQUARE (FIRST);
;; S : constant INTEGER := FIRST;
;; return S * S;                              <-- 2nd inline expansion
        XWMUL    0,19.,3           ;; S
        XWSTA    0,17.,3           ;; S
;; return S * S;                              <-- 3rd inline expansion
        XWMUL    0,17.,3           ;; S
        XWSTA    0,14.,3           ;; SECOND
        WRTN
;; end

;; function SQUARE (S : INTEGER) return INTEGER is
;; begin
;; return S * S;
        XWLDA    0,@-12.,3                 inlined
        XWMUL    0,@-12.,3                 subprogram
        XWSTA    0,-8.,3
        WRTN
;; end
```

Pragma INLINE (example continued)

.SR Instructions Corresponding to WITHOUT_INLINE.ADA

```
;; begin
;; FIRST := SQUARE (2);
        LPEF     L3              <-- push effective address [L3]=2

        LCALL    L2,1,1          <-- 1st call to SQUARE
        XWSTA    0,12.,3
;; SECOND := SQUARE (SQUARE (FIRST));
        XWSTA    0,17.,3
        XPEF     17.,3           <-- push effective address [17]=4

        LCALL    L2,1,1          <-- 2nd call to SQUARE
        XWSTA    0,19.,3

        XPEF     19.,3           <-- push effective address [19]=16

        LCALL    L2,1,1          <-- 3rd call to SQUARE
        XWSTA    0,14.,3
        WRTN
;; end

;; function SQUARE (S : INTEGER) return INTEGER is
;; begin
;; return S * S;
L2:
        XWLDA    0,@-12.,3
        XWMUL    0,@-12.,3                          called
        XWSTA    0,-8.,3                            function
        WRTN
;; end

L3:     2
        .END
```

---

pragma INTERFACE

Specifies another language (and calling conventions) for interfacing with an ADA program.

Format: pragma INTERFACE (language name, subprogram name);

language name — is the name of the language of the called subprogram.

subprogram name — is the name of the called subprogram, which you have called earlier in this declarative part.

Description:

Pragma INTERFACE must appear as a declarative item in the declarative part or package specification of the Ada unit that does the calling. The subprogram name you specify as a pragma argument must be on you have delared warlier in the same declarative part or package specification.

Pragma INTERFACE allows you to call program units written in other languages. A specification for the named subprogram must be written in ADA. Pragma INTERFACE allows the body of the subprogram to be written in the named language.

Pragma INTERFACE may appear at the place of a declarative item in a declarative part of a subprogram or package. The subprogram mentioned must be a subprogram declared earlier in the same declarative region.

The interface runtime requires your program to include the following pragma LOAD statements (in this order), which ensure that corresponding objects are loaded in the correct order:

```
pragma LOAD ("ADE_ROOT?:RUNTIMES:INTERFACE_LRT_TRIGGGER");
pragma LOAD ("LANG_RT.LB");
```

[LANG_RT.LB must be accessible via one of the system provided file access methods (searchlist, links, etc.]

The IMPORT command does this for you automatically, and you should use it to import routines written in F77, C, or PASCAL.

ADA currently supports the calling of subprograms written in F77, PASCAL, C, MASM and ASSEMBLY. You can call any language that obeys the common calling conventions of DG languages, but will receive a compiler warning that the language is not explicitly supported.

The Ada interface traps any runtime error in the called routine and raises the PROGRAM_ERROR exception in the Ada caller. Before Revision 2.30 errors in the called (non-Ada) routine would cause a stack traceback and the program would terminate.

The Ada interface suspends Ada tasking during the call to the non-Ada subroutine.

General Notes:
----

* The ADA compiler now generates error and/or warning messages relating to foreign calls.

* Characters within constructs are packed according to DG alignment requirements for the called language.

* Booleans are passed 1 per word, packing is not done for arrays or records.

* Return values are not checked for validity.

* Procedure and Function calls to other languages do not support type conversions. Type conversions must be done explicitly.

* Should an error occur within a foreign language PROGRAM ERROR will be raised on return to the ADA program.

* Passing of ACCESS types is generally allowed, however the programmer should exercise caution when changing ADA data structures. A description of 2.40 data formats is provided at the end of this document. Data General may change these formats in a future revision. Programs which depend on the currently used data formats should be vigorously tested after receiving each new revision.

* When foreign subprograms are called, exception handling is performed by LANG_RT. If a foreign subprogram has an error, that error will be propagated to the calling ADA program as a PROGRAM_ERROR.

* Foreign subprograms MUST be in the same ring as the calling ADA program.

General Notes: (continued)

* I/O can be done from foreign routines, however it is the users responsibility to PRAGMA LOAD all necessary runtime objects. Alternately the programmer can use the template facility provided by ADALINK.

* The foreign code interface does not support Ada unconstrained types for any of the languages.

* All appropriate LB's and OB's must be loaded into Ada programs calling foreign programs. The IMPORT function only ensures that the OB containing your function and LANG_RT are pragma loaded. If the foreign code requires additional runtime support, such as MULTITASKING.OB, the names of all necessary OB's and LB's need to be added to <interface_package>_B which is created by IMPORT or by ADALINK templates.

MASM or ASSEMBLY

The MASM and ASSEMBLY options provide the standard ADA calling conventions. If either is specified the called program (which may or may not be MASM or ASSEMBLY) is expected to follow ADA calling conventions and to know how ADA data structures will be formatted.

F77

The F77 option provides is supported as follows:

F77 Data Type	Ada Data Type
INTEGER*4	INTEGER
INTEGER*2	SHORT_INTEGER
REAL*4	FLOAT
REAL*8	LONG_FLOAT
CHARACTER*1	CHARACTER
CHARACTER*N	STRING(1..N)
ARRAY	ARRAY

NOTES:

Array elements must be of a supported scalar type.
Scalar parameters are passed copy-in copy-out.
One dimension arrays mode IN OUT are passed by reference.
N Dimension arrays obey copy-in copy-out rules.

C

C is supported as follows:

C Data Type	Ada Data Type
SHORT_INT	SHORT_INTEGER
LONG_INT	LONG_INTEGER
SHORT_FLOAT	FLOAT
LONG_FLOAT	LONG_FLOAT
CHARACTER	CHARACTER
POINTER	ACCESS
ENUMERATION	ENUMERATION
ARRAY OF CHARACTER	STRING
ARRAY	ARRAY
STRUCTURE	RECORD

NOTES:
C calling conventions specify pass by value. Therefore only IN mode is allowed for scalar parameters and structures. The call interface enforces pass by value for arrays.

PASCAL

The PASCAL option is supported as follows:

PASCAL Data Type	Ada Data Type
SHORT_INTEGER	SHORT_INTEGER
LONG_INTEGER	INTEGER
REAL	FLOAT
DOUBLE_REAL	LONG_FLOAT
BOOLEAN	BOOLEAN
CHAR	CHARACTER
ENUMERATION	ENUMERATION
POINTER	ACCESS
ARRAY	ARRAY
PACKED ARRAY OF CHAR	STRING
RECORD	RECORD

NOTES:
not supported: RECORD VARIANTS, SET, FILE .
COPY IN , COPY OUT will be used except for a one dimensional array of scalars which will be passed by reference for IN OUT.

PL/1

The PL/1 option is supported as follows:

PL/1 Data Type	Ada Data Type
FIXED BINARY (15)	SHORT_INTEGER
FIXED BINARY (31)	INTEGER
FLOAT BINARY (21)	FLOAT
FLOAT BINARY (53)	LONG_FLOAT
POINTER	ACCESS
ARRAY	ARRAY
RECORD	RECORD

NOTES: PL1 is not explicitly supported, however the above data types may be used IF all data follows standard LANG_RT alignment/space characteristics. Specifying PL1 as the interface language will result in a compilation warning. COPY IN , COPY OUT will be used except for a 1 Dim array of scalars which will be passed by reference for IN OUT.

pragma LIST

Suspends or resumes the compiler listing file ouput.

Format: pragma LIST (ON | OFF);

Description

The compiler always produces a listing (.LST) file unless

- you include the /ERRORS switch on the ADA command line and the compilation units contains no errors;
- the compilation unit includes the statement, "pragma LIST (OFF);"

"Pragma LIST (OFF);"	suspends .LST output file listings of the compilation.
"Pragma LIST (ON);"	resumes .LST output.

Example:

```
procedure MAIN is
   type MEMBERS is private;
   procedure SORT (LIST: in out MEMBERS);
   function HEAD (L: LIST) return MEMBERS;
   ...
   pragma LIST (OFF);                            <-- suspends
   type MEMBERS is
   ...
   end MEMBERS;
   pragma LIST (ON);                             <-- resumes
begin
...
end MAIN;
```

---

pragma LOAD

Includes non-Ada object files in the linked program file for this compilation.

Format: pragma LOAD ("object_file_pathname");

object_file_pathname — is the STRING literal (in quotes) that denotes the full pathname of the non-Ada object file you want to load. You need not include the .OB filename extension.

Description

Pragma LOAD allows you to include foreign (non-Ada) object files in your program. You can use it in conjunction with pragmas INTERFACE and ENTRY_POINT to enable Ada procedures to call non-Ada subprograms.

The Ada Linker includes the named object file when it builds the Ada program (.PR) file.

Pragma LOAD must appear at the head of a compilation for a body. When using pragma LOAD with compilation subunits, always specify the "/READ_SUBUNITS" switch on the ADALINK command line. If you omit that switch, you may receive this error message from the Linker:

"Can't get [body] tree for <program_unit_name>"

Example:

```
pragma LOAD ("SEVEN_UP");                                 <-- named .OB file
with TEXT_IO; use TEXT_IO;                                    is in the
procedure ADA_CALLS_PL1 is                                    current
   procedure SEVEN_UP (X: out INTEGER);                       directory.
   pragma INTERFACE (PL1, SEVEN_UP);
   pragma ENTRY_POINT (SEVEN_UP, "SEVEN_UP");
   N : INTEGER;
begin
   SEVEN_UP (N);
   PUT (N);
end ADA_CALLS_PL1;
```

pragma MAIN

Indicates that a subprogram unit is a main program.

Format: pragma MAIN;

Description

Pragma MAIN designates the main subprogram unit. Place pragma MAIN immediately after the subprogram you want to be the main subprogram.

Example:

```
procedure TEST is                        <-- main procedure

   procedure FIRST is
   ...
   end FIRST;

   procedure SECOND is
   ...
   end SECOND;
begin
   ...
end TEST;
pragma MAIN;                             <-- pragma
```

Another way to distinguish the main subprogram in a compilation unit is to use the "/MAIN_PROGRAM=name" switch on the ADA command line. For example, this command

```
-) ADA/MAIN_PROGRAM=TEST TEST
```

compiles the procedure TEST, located in the source file "TEST.ADA" as a main program. For more information about the ADA command line, see "ADE User's Manual", Chapter 3.

---

pragma MAX_TASKS

Specifies the maximum number of Ada tasks you want active simultaneously.

Format: pragma MAX_TASKS (n)

n is an INTEGER value greater than 0.

Description

Pragma MAX_TASKS specifies the maximum number of Ada tasks that can be active concurrently. If you do not specify the number, the system gives you a maximum of 50.

This pragma must appear at the head of a compilation. It applies to all units in the compilation.

Example:

```
pragma MAX_TASKS(40);                             <-- pragma
package body TASKS is
   ...
   task ONE is ...;
   task TWO is ...;
   task type THREE_TO_FORTY is ...;
   type REMAINING_TASKS is
      array (3..40) of THREE_TO_FORTY;
   MULTI_TASKS : REMAINING_TASKS;
   ...
end TASKS;
```

You can also specify the maximum number of tasks by using the "MAX_TASKS=n" switch on the ADALINK command. For example,

```
-) ADALINK/MAX_TASKS=40  object_filename
```

If you specify a maximum number of Ada tasks with both a pragma and a switch, the pragma takes precedence. For more information, see "ADE User's Manual", chapter 4.

---

pragma MV_ECS

Specify the use of the Data General MV External Calling Sequence.

To optimize code quality, the Ada compiler does not always generate code which conforms with the Data General MV External Calling Sequence (ECS). In some cases, however, you will need to indicate to the compiler that MV ECS is necessary. Subroutines which meet any of the following criteria must use MV ECS:

* MACHINE_CODE subroutines with formal arguments

* subroutines called from other DG languages

* subroutines which may be called from outer rings

Use pragma MV_ECS to indicate the need for use of MV ECS. The syntax for this pragma is:

```
pragma MV_ECS( unit_name [, unit_name...]);
```

For example:

```
pragma MV_ECS( subprog1, subprog2, subprog3 );
```

pragma PAGE

Begins a new page in the compiler output listing file.

Format: pragma PAGE;

Description

The compiler produces a listing (.LST) file unless

1. You have included the /ERRORS switch on the ADA command line and the compilation units contains no errors;

2. The compilation unit includes the statement, "pragma LIST (OFF);"

If the compiler is producing a listing of the compilation, pragma PAGE causes the text following the pragma to appear on a new page.

Example:

```
procedure FIRST is
...
end FIRST;

pragma PAGE;              <-- begin new .LST pages
procedure SECOND is          /
...                         /
end SECOND;                /
                          /
pragma PAGE;             /
...
```

pragma PRIORITY

Specifies the priority of a task, a task type, or a main program unit.

Format: pragma PRIORITY (n);

n is an INTEGER value from 1 to 10. Lower values indicate lower priorities.

Description

You assign tasking priorities by including pragma PRIORITY within appropriate task specifications, or within the outermost declarative part of a main program.

You can assign each task or task type a priority, but the assignment is optional. Assigning tasking priorities directs the system how to handle competing tasks. When more than one task is eligible for execution at a time, the system executes them in the order you specify with PRIORITY pragmas.

Ready tasks are queued first by priority number and within priority by order of their occurrence in the source file (FIFO). You may assign each task, task type, or main program only one priority. If you assign more than one priority the system recognizes only the first assignment and ignores the others.

The default task priority is 5.

Example:

```
procedure OUTER is
   pragma PRIORITY (4);            <-- main procedure priority
   ...                                 assignment
   task type TASK_TYPE is
      pragma PRIORITY (7);         <-- task priority assignment
      ...
   end TASK_TYPE;
   ...
end OUTER;
```

pragma SUPPRESS

Suppresses specified runtime checks.

Format: pragma SUPPRESS (check_identifier [, [ON=>] name]);

check_identifier	names the check you want to suppress; check identifier names are listed below.
name	is the name of a type, subtype, object task unit, generic unit, or subprogram.

Description

To suppress certain runtime checks, place pragma SUPPRESS in the declarative part of a program unit or block, or immediately within a package specification.

For a program unit or block, check suppression extends from the occurrence of the pragma to the end of the declarative region associated with that program unit or block.

For a package, check suppression extends to the end of the scope of the specified "ON =>" entity. You must declare that entity immediately within the package specification.

The following table shows the extent of check suppression for each named entity.

Check suppression for	Extends over
An unnamed entity (name omitted)	The remaining declarative region
An object	All operations of the object
An object of the base type or subtype	All operations of the object
A task or task type	All activations of the task(s)
A generic unit	All instantiations of the generic
A subprogram	All calls of the subprogram

pragma SUPPRESS (continued)

Although it is better programming practice to have runtime exceptions automatically raised, you can suppress them in order to decrease runtime overhead. When you suppress runtime checks, you effectively turn off certain program exceptions. Of course, if an error situation arises after you have suppressed a check, your compiled program will be erroneous. The following table shows which program exceptions you turn off when you suppress checks:

ppression of his checkname	Turns off this exception	When program detects this runtime error
CCESS_CHECK	CONSTRAINT_ERROR	Selection or indexing applied to an object with a null value.
ISCIMINANT CHECK	CONSTRAINT_ERROR	Violation of discriminant constraint.
DEX CHECK	CONSTRAINT_ERROR	Out-of-range index values.
ENGTH_CHECK	CONSTRAINT_ERROR	Wrong number of index components.
ANGE CHECK	CONSTRAINT_ERROR	Values exceed range constraint, or type is incompatible w/constraint.
IVISION_CHECK	NUMERIC_ERROR	Division, rem, or mod by zero.
ERFLOW CHECK	NUMERIC_ERROR	Operation result exceeds implemented range.
LABORATION CHECK	PROGRAM_ERROR	Attempt to call a unit before it is elaborated.
TORAGE_CHECK	STORAGE_ERROR	Over-allocation of memory space.

ample:

```
procedure MAIN is
   type COLOR is (RED, BLACK);
   type TABLE is array (1..8, 1..8) of COLOR;
   pragma SUPPRESS (INDEX_CHECK, ON=> TABLE);   <-- suppression extends
   X, Y : INTEGER;                                  to all type TABLE
   BOARD : TABLE;                                   operations in MAIN.
begin
   ...
   BOARD (X, Y) := RED;                         <-- X+Y may not be in the
   ...                                              range 1..8, but that
end;                                                check is suppressed.
```

---

pragma TASK_STORAGE_SIZE

Specifies the amount of heap storage space to allocate for task stacks.

Format: pragma TASK_STORAGE_SIZE (n);

n is the total number of 2-byte words you want to allocate for all active task stacks. n can be any INTEGER value, but only values greater than -1 have an effect.

Description

Pragma TASK_STORAGE_SIZE allows you to reset the amount of heap space to allocate for all task stacks. The amount of space you specify should exceed the amount of storage you need at one time for all active tasks. By default, the system allocates 128K words.

You can also use the "/TASK_STORAGE_SIZE=n" switch on the ADALINK command line to control the maximum heap space allocated to active task stacks. If you use both the pragma and the command switch, the pragma takes priority.

The pragma must appear at the head of a compilation; it applies to the entire compilation unit.

Resetting MTOP

If you set TASK_STORAGE_SIZE to a value greater than the current virtual address space will allow, you must reset the maximum virtual address space by specifying the value of MTOP. MTOP defines the maximum virtual address for a program. Use the "/MTOP=n" switch on the ADALINK command, where n specifies how many megabytes your program will require. The default value of MTOP is 1 Mbyte.

For example, this command resets MTOP to 20 Mbytes:

```
-) ADALINK/MTOP=20  object_file
```

pragma TASK_STORAGE_SIZE (continued)

Individual Task Storage

By default, the system allocates 2048 words for each active task stack. If you require a larger or smaller stack for a particular task or task type stack, use the STORAGE_SIZE representation clause. For example, the following clause directs the compiler associate task type BIG with a stack of size N:

```
for BIG'STORAGE_SIZE use N;
```

The minimum stack size that you can specify is 512 words.

Example:

```
pragma TASK_STORAGE_SIZE(56_000)       <-- value exceeds
procedure MAIN is                          storage required
...                                        for all parallel
    task ONE is ...;                       tasks
    for ONE'STORAGE_SIZE use 1_000;

    task TWO is ...;
    for TWO'STORAGE_SIZE use 2_000;
    ...

    task TEN is ...;
    for TEN'STORAGE_SIZE use 10_000;

end MAIN;
```

THE PACKAGE SYSTEM

The predefined library package called SYSTEM declares the following types, subtypes, and objects which characterize system-specific values and operations applicable to those values.

This section describes the type, subtype, and object declarations that comprise the package SYSTEM in ADE. The following lists the specification of this package, which is outlined in section 13.7 of the LRM:

```
package SYSTEM is

    type ADDRESS is new INTEGER;
    type NAME is (MV);
    SYSTEM_NAME  : constant := NAME := MV;
    STORAGE_UNIT : constant := 16;
    MEMORY_SIZE  : constant := 2 ** 29;

    MAX_INT      : constant := (2**30) - 1 + (2**30);
    MIN_INT      : constant := -MAX_INT - 1;
    MAX_DIGITS   : constant := 15;
    MAX_MANTISSA : constant := 30;
    FINE_DELTA   : constant := 2.0 ** (-30);
    TICK         : constant := 0.1;

    subtype PRIORITY is INTEGER range 1..10;

end SYSTEM;
```

Type or Constant	Defined as	Explanation
ADDRESS	INTEGER	Address clauses and attributes (P'ADDRESS) return objects of the derived type ADDRESS.
NAME	MV	The enumeration type NAME declares one object: the literal MV.
SYSTEM_NAME	MV	SYSTEM_NAME is an object of type NAME and is initialized to MV.

Package SYSTEM (continued)

SYSTEM Type or Constant	Defined as	Explanation
STORAGE_UNIT	16	Denotes the number of bits per storage unit.
MEMORY_SIZE	2**29	Denotes the number of available storage units.
MAX_INT	(2**30)-1+(2**30) = 2_147_483_647	Denotes the highest value of predefined INTEGER types.
MIN_INT	-MAX_INT - 1 = -2_147_483_648	Denotes the lowest (most negative) value of pre-defined INTEGER types.
MAX_DIGITS	15	Denotes largest number of significant decimal digits in a floating-point constraint.
MAX_MANTISSA	30	Denotes the largest allowed number of binary digits in the mantissa of model numbers of a fixed-point subtype.
FINE_DELTA	2.0**(-30)	Denotes the smallest delta allowed in a fixed-point constraint that has the range constraint -1.0..1.0.
TICK	0.1	Denotes the basic clock period, in seconds.
PRIORITY	1..10	Subtype PRIORITY (base type INTEGER) declares the range of values you can use on pragma PRIORITY statements.

Representation Clauses

This section describes the use of representation clauses in the ADE. You may use representation clauses for either of two purposes:

1. To specify a more efficient representation of data in the underlying machine.

2. To communicate with features outside the domain of the Ada language; for example, peripheral hardware.

The Ada language provides four classes of representation clauses, as the following table describes.

Clause Class	Specifies
Length clause	The amount of storage you want associated with a type.
Enumeration representation	The internal codes for the literals of an enumeration type.
Record representation	The storage order, relative position, and size of record components.
Address clause	The required address in storage for an entity.

Length Clause

You can use the 'STORAGE_SIZE attribute only for reserving storage for activation of a task or a task type. For example:

```
BITS     :constant:=1;
BYTES    :constant:=8*BITS;
KBYTES   :constant:=1024*BYTES;

task MONITOR is ...;

for MONITOR'STORAGE_SIZE use 4*KBYTES;      <-- length
                                                clause
```

Representation Clauses (continued)

Enumeration Representation

We now support enumeration representation clauses as specified in the language reference manual (13.3:4). All enumeration literals must be provided with distinct static integer codes; the sequence of integer codes specified for the enumeration type must be montonicly increasing.

There are, however, two restrictions: the range of internal codes is 0 .. SHORT_INTEGER'LAST, and enumeration types with representation clauses are not allowed as an index type of an array type definition (3.6:2).

Change of Representation

Declaration of a derived type in order to specify an alternative representation is supported as specified in the language reference manual. A change of representation can be accomplished using as explicit conversion.

Operations of Discrete Types

Discrete attributes 'POS, 'VAL, 'SUCC, and 'PRED with a discrete type or subtype whose base type is enumeration with a representation clause, will involve additional runtime overhead (as implied in the language reference manual 13.3:6) if the argument is non-static. If the argument is static there is no additional runtime overhead.

This additional runtime overhead is a mapping from potentially non-contiguous internal codes to position numbers, and vice versa.

The forementioned attributes with a discrete type or subtype other than enumeration with a representation clause are unaffected.

Type Conversions

Explicit conversions between enumeration types where either base type has a representation clause will involve additional runtime overhead (as explained in section 3.5.5) if the argument is non-static. If the argument is static there is no additional runtime overhead.

Case Statements

If the base type of the case statement expression is an enumeration type with a representation clause, the resulting code will be optimized with respect to space rather than time (i.e. sequential compares-never-branch tables will be used to determine the proper case statement alternative).

Case statements with types other than enumeration with a representation clause are unaffected.

Loop Statements

Loop statements with a for iteration scheme where the base type of the loop parameter is enumeration with a representation clause will involve additional runtime overhead. (See the 'SUCC 'PRED explanation in section 3.5.5.)

Loop statements with a for iteration scheme where the base type of the loop parameter is not enumeration with a representation clause are unaffected.

Parameter Associations

If the actual parameter has the form of a type conversion where the base type of the type mark is enumeration with a representation clause, there will be runtime overhead in performing the conversion (see 4.6). The amount of overhead depends on the parameter (6.4.1:4), and whether the expression given as the operand is static.

Explicit conversions as actual parameters with types other than enumeration with a representation clause are unaffected.

Record Representation

Representation of record types in ADE is the same as in standard Ada, with certain restrictions. Specifically, you cannot use record representation clauses to specify alignment and component locations for the following kinds of record types:

- Record types with discriminants.

- Record types with variant parts.

- Record types with arrays.

When specifying component storage, you can overlap only one 16-bit word boundary. You cannot specify the storage for composite, FLOAT, or LONG_FLOAT components. The compiler automatically determines the required storage for components of those restricted types. You can specify storage for all the remaining component types, as the language permits.

===

The following example shows a valid record representation specification:

```
type IUFL is
   record
      RETURN_FLAGS      : INTEGER range 0 .. 15
      TERMINATION_FIELD : INTEGER range 0 .. 7;
      PROCESS_ID        : INTEGER range 1 .. 255;
   end record;

for IUFL use
   record
      RETURN_FLAGS      at 0 range 0 .. 4;
      TERMINATION_FIELD at 0 range 5 .. 7;
      PROCESS_ID        at 0 range 8 .. 15;
    end record;

for IUFL'SIZE use SYSTEM.STORAGE_UNIT;
```

These component clauses specify the order, position, and size of IUFL fields relative to the start of the IUFL record. More specifically, they ensure IUFL fields match the following structure of the ?IUFL offset (user flag word) in a ?IREC system call:

0 4	5 7	8 15
?RETURN flags	Termination field	PID of terminating process

Structure of offset ?IUFL (user flag word)

In addition, ADE does not allow components to overlap storage boundaries. That is, record fields cannot cross more than one 16-bit word boundary.

Address Specification

In the ADE, you can use address clauses to specify only the fixed internal names for Ada subprograms. Wherever possible, use pragma ENTRY_POINT to assign internal names to subprograms.

The compiler recognizes address clauses of this form only:

```
for SUBPROGRAM_NAME use at ####;
```

where #### is a STANDARD.ADDRESS value in the range 3001 to 5000. The compiler treats #### as a label name of the form E#### for the entry point "SUBPROGRAM_NAME."

For example, the following program segment uses an address clause to assign the fixed internal name "E3001" to the subprogram "SEVEN_UP":

```
procedure SEVEN_UP(M:out INTEGER);
for SEVEN_UP use at 3001;                      <--- address
procedure SEVEN_UP (M: out INTEGER) is              clause
   ...
end SEVEN_UP;
```

Code written in assembly language can call Ada code through the internal label. Label values 0 through 3000 are reserved for runtime routines, the Debugger and the Kernel. Label values 3001 through 5000 are available for your use.

UNCHECKED PROGRAMMING

The ADE implements the predefined generic library subprograms UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSIONS. The following paragraphs explain how to use those subprograms.

Procedure UNCHECKED_DEALLOCATION

You can use the generic procedure UNCHECKED_DEALLOCATION to explicitly deallocate dynamic objects that are designated by values of access types.

To deallocate dynamic objects explicitly, your program must instantiate this procedure for a particular object and access type. In the program body, a call to the instantiated procedure specifies the dynamic object as a parameter. When that call is executed, the specified object is deallocated and its value is set to null. The following example shows how this works.

Example:

```
 with UNCHECKED_DEALLOCATION;
 package TREE_LABELER is

    type LABEL_TYPE is private;
    type NODE;
    type TREE is access NODE;
    type NODE is record
        LABEL : LABEL_TYPE;
        LEFT  : TREE;
        RIGHT : TREE;
    end record;

    procedure DISPOSE is new UNCHECKED_DEALLOCATION (NODE, TREE);
    procedure LABEL_ROOT (LABEL : in LABEL_TYPE;
                          ROOT  : in out TREE;
                          LABELLED_TREE : out TREE);
end TREE_LABELER;

package body TREE_LABELER is

    procedure LABEL_ROOT (LABEL         : in LABEL_TYPE;
                          ROOT          : in out TREE;
                          LABELLED_TREE : out TREE);
    ROOT1, ROOT2 : NODE;
    begin
        ...
        DISPOSE (ROOT1);
        ...
    end LABEL_ROOT;
end TREE_LABELER;
```

UNCHECKED_DEALLOCATION (continued)

In this example, the call to the procedure DISPOSE deallocates the dynamic object designated by the access value ROOT1, and resets ROOT1 to null. Note, however, that if the enclosing procedure uses the other access value, ROOT2, to designate the same object as ROOT1, this will result in an erroneous program since the object no longer exists. You must be wary of similar "dangling references" when using the procedure UNCHECKED_DEALLOCATION.

Function UNCHECKED_CONVERSION

The generic function UNCHECKED_CONVERSION allows you to return the value of an IN parameter as a value of a target type. The actual bit pattern corresponding to that parameter value does not change.

The function UNCHECKED_CONVERSION is a unit in the ADE SYSTEM library. Here is the visible part of that function:

```
generic
type SOURCE is limited private;
type TARGET is limited private;
function UNCHECKED_CONVERSION (S : SOURCE) return TARGET;

function UNCHECKED_CONVERSION (S : SOURCE) return TARGET is
   pragma SUPPRESS (RANGE_CHECK);
begin
   return S;
end UNCHECKED_CONVERSION;
```

UNCHECKED_CONVERSION (continued)

For instantiations of this generic function, types SOURCE and TARGET must be of the same class and the same length. Note: the current revision of ADE does not allow SOURCE and TARGET to be array types.

Example:

```
with UNCHECKED_CONVERSION, ALPHA;
package BETA is
   type TEST_NAME is private;
   type DATA is record
          IS_VALID : BOOLEAN;
          TEST_OBJECT : TEST_NAME;
       end record;
    ...
    function CONVERT_TO_BETA_DATA is new
       UNCHECKED_CONVERSION (ALPHA.INFO, DATA);

    function CONVERT_FROM_BETA_DATA is new
       UNCHECKED_CONVERSION (DATA, ALPHA.INFO);
    ...
end BETA;
```

For more information about unchecked conversions, see LRM, Section 13.10.

CHARACTERISTICS OF ADE I/O PACKAGES

The standard input and output files in TEXT_IO correspond to the AOS/VS generic files @INPUT and @OUTPUT respectively. For more information about AOS/VS generic files, see the DGC manual, "Learning to Use Your AOS/VS System."

The maximum value for TEXT_IO.COUNT and TEXT_IO.FIELD is SYSTEM.MAX_INT.

The FORM parameter of the TEXT_IO.OPEN procedure is currently not used.

Type TEXT_IO.FILE_TYPE is an access type.

For more information about input/ouput operations in the ADE, see "ADE User's Manual", chapter 6.

MAXIMUM SIZE LIMITS IN ADE

The package LIMITS specifies the following absolute limits on the use of Ada language features:

Compilation step	Language Feature	Maximum or amount
Syntax parsing	Length of identifiers	120
	Length of line	120
Semantics checking	Discriminants in constraint	256
	Associations in record aggregate	256
	Fields in record aggregate	256
	Formals in generic	256
	Nested contexts	250
Generating machine code	Indices in array aggregate	128
	Parameters in call	128
	Nesting depth of expressions	100
	Nesting depth of inlined expressions	100
	Nesting depth of packages with tasks	100

Summary of Implementation-Dependent Real Type Attributes

Float Types:

T'MACHINE_RADIX	= 16
T'MACHINE_MANTISSA	= number of T'MACHINE_RADIX (hex) digits in mantissa. Value is 6 for FLOAT, 14 for LONG_FLOAT
T'MACHINE_EMAX	= 63 maximum exponent for MV floating types, base 16
T'MACHINE_EMIN	= -64 minimum exponent for MV floating types, base 16
T'MACHINE_ROUNDS	= TRUE
T'MACHINE_OVERFLOWS	= TRUE
T'SAFE_EMAX	= 252 formula: log2 (T'MACHINE_RADIX) * T'MACHINE_EMAX
T'SAFE_SMALL	= 2.0 ** (-T'SAFE_EMAX - 1)
T'SAFE_LARGE	= 2.0 ** T'SAFE_EMAX * (1.0 - 2.0 ** (-T'BASE'MANTISSA)

Fixed Types:

T'MACHINE_ROUNDS	= TRUE
T'MACHINE_OVERFLOWS	= TRUE
T'BASE'SMALL	= T'SMALL
T'BASE'MANTISSA	= 31 (Same as SYSTEM.MAX_MANTISSA)
T'SAFE_SMALL	= T'BASE'SMALL
T'SAFE_LARGE	= T'BASE'LARGE also = (2 ** T'BASE'MANTISSA - 1) * T'BASE'SMALL

Notes:

* All fixed point numbers are stored in 32-bit INTEGERs.

* Floating point types requiring five or less digits of precision are stored in FLOAT; those requiring six to 14 digits are stored in LONG_FLOAT.

* FLOAT and LONG_FLOAT use one bit for the sign and seven bits for the exponent (of 16) in excess-64 notation. FLOAT has 24 bits available for the mantissa; LONG_FLOAT has 56.

* For FLOAT and LONG_FLOAT, the smallest representable number in the MV architecture is T'MACHINE_RADIX ** (-T'MACHINE_EMIN -
  16 ** (-65)
  (that is, .10000000000000#16# ** 16 ** (-64))

* For FLOAT and LONG_FLOAT, the largest representable number in the MV architecture is
```
     (1.0 - T'MACHINE_RADIX ** (-T'MACHINE_MANTISSA - 1) *
     (T'MACHINE_RADIX ** T'MACHINE_EMAX)
```
  (that is, .FFFFF#16# * 2 ** (63) for FLOAT, and .FFFFFFFFFFFFFF#16# * 2 ** (63) for LONG_FLOAT).

-- End of Appendix F.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	119(A),(1)
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	119(A),(2)
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	60(A),(3),59(A)
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	60(A),(4),59(A)
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	117(0),(298)
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	114(0),(69.0E1)

$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	60(A)
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	59(A),(1)
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	100(" ")
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	X}!@#$^&~Y
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	XYX*
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	4194303.998
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	4194305.000
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BAD-CHARACTER*^

$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	MUCH_MUCH_TOO_LONG_NAME_FOR_A_FILE
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-4194305.000
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-4194305.000
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	120
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(2:),113(0),(11:)

$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(16:),112(0),(F.E)
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	120(A)
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_ INT.	-2127283648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	not supported
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	8#37777777776#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 28 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

B28003A: A basic declaration (line 36) wrongly follows a later declaration.

E28005C: This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ARG.

C34004A: The expression in line 168 wrongly yields a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

C35502P: Equality operators in lines 62 & 69 should be inequality operators.

A35902C: Line 17's assignment of the nominal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may in fact raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

C35A03E, & R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard doesn't support this assumption.

C37213H: The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

C37213J: The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

C37215C, E, G, H: Various discriminant constraints are wrongly expected to be incompatible with type CONS.

C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

C41402A: 'STORAGE_SIZE is wrongly applied to an object of an access type.

C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

C45614C: REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

E66001D: Wrongly allwos either the acceptance or rejection of a parameterless function with the dame identifier as an enumeration literal; the function must be rejected (see Commentary AI-00330).

A74106C, C85018B, C87B04B, CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur re lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere).

BC3105A: Lines 159..168 are wrongly expected to be illegal; they are legal.

AD1A01A: The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

CE2401H: The record aggregates in lines 105 & 117 contain the wrong values.

CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.